Efficient Distribution of Full-Fledged XQuery

Ying Zhang Nan Tang Peter Boncz
CWI, Amsterdam, The Netherlands
{Y.Zhang, N.Tang, P.Boncz}@cwi.nl

***Abstract*— We investigate techniques to automatically decompose any XQuery query into subqueries, that can be executed near their data sources; i.e., function-shipping. In this scenario, the subqueries being executed remotely may have XML node-valued parameters or results, that must be shipped in some way. The main challenge addressed here is to ensure that the decomposed queries properly respect XML *node identity* and preserve *structural properties*, when (parts of) XML nodes are sent over the network, effectively copying them.**

We start by precisely characterizing the conditions, under which *pass-by-value* parameter passing causes semantic differences between remote execution of an XQuery expression and its local execution. We then formulate a conservative strategy that effectively avoids decomposition in such cases. To broaden the possibilities of query distribution, we extend the pass-by-value semantics to a *pass-by-fragment* semantics, which keeps better track of node identities and structural properties. The pass-by-fragment semantics is subsequently refined to a *pass-by-projection* semantics by means of a novel runtime XML projection technique, which safely eliminates most semantic differences between the local and remote execution of an XQuery expression, and strongly reduces message sizes.

The proposed techniques are implemented in XRPC, a simple yet efficient XQuery extension that enables function-shipping by adding a Remote Procedure Call mechanism to XQuery. Experiments on *MonetDB/XQuery* establish the performance potential of our XQuery decomposition techniques.

I. Introduction

In this paper, we study ways to decompose any XQUERY query that consults multiple XML documents residing on multiple peers into subqueries that can be executed on those peers, in other words, function shipping. In principle, we do not want to restrict the form of these queries in any significant way: the full-fledged XQUERY language is our decomposition starting point. The goal of this paper is to be able to exploit the computational power of (heterogeneous) XQUERY engines on the Web to jointly execute XQUERY queries.

Note that XQUERY already allows queries over distributed sources through its support for W3C standards, in particular, the ability to open *any* document on the Web through its fn:doc(URI) built-in function. However, the execution model implied by those W3C standards (such as HTTP) is data shipping, i.e., the transportation of a full XML document from a remote to the querying peer. This means that all query execution happens locally, i.e., at the query originator. It is well known that in many cases this is suboptimal.

Decomposing queries to address multiple data sources is a well-studied optimization problem in relational [28], object-oriented [13], [17], and semi-structured databases [24], [25]. While it is natural (and true) to assume that many of the existing techniques can be carried over, the XML data model and the XQUERY language introduce a number of particular challenges not met elsewhere, that revolve around XML node identity and structural (rather than value-based) relationships between nodes. Previous work on distributed XML [6], [7], [26] only focused on a restricted set of XQUERY queries, and did not address the problem of transparent query decomposition, such that these challenges did not play a role.

Shipping XML Messages. Without loss of generality, we view the subexpressions to be executed by remote peers as XQUERY functions, that may have parameters and produce a result. During *remote* function execution, the calling peer (e.g., query originator) will send a request message containing parameters to a remote peer, which executes the subexpression, and sends back a response message containing the result. To illustrate the challenges of distributing XQUERY, yet preserving XML node identity, consider a subexpression f($a,$b) with two parameters $a and $b of the type node(), that is executed remotely. Complications may arise, for instance, if the subexpression f() tests structural XML relationships among its parameters, such as "$a/parent::b is $b". It therefore depends on the characteristics of the subexpressions f as well as on the way parameters are marshaled in and out of the network messages whether f will behave correctly, that is, identical to local XQUERY execution (note that blindly copying all parameters into the message does not work in this example).

When XML nodes must be shipped over the network, this means that unless one chooses to ship the entire underlying XML document in order to preserve all structural relationships (which defeats the purpose of function shipping), pieces/snippets of the underlying XML document must somehow be copied into the messages, changing the "holistic" structural properties and identity of the nodes, which may affect the semantics of XQUERY execution on such shipped nodes. Naively, when shipping a node, one would ship its descendants (XML subtree), but other solutions are also possible, and will in fact be proposed in this paper (in particular the idea to use XML projection techniques).

XRPC. While our problem statement covers distributed XQUERY in general, this research stems from the particular context of the XRPC project [30], [31]. XRPC adds the concept of Remote Procedure Call to XQUERY by introducing a single new statement: execute at {Expr}{FunApp(ParamList)}, where Expr specifies the URI (either a constant or a computed one) of the peer, on which FunApp() will be executed. It also supports an xrpc:// scheme in the URI parameter of the fn:doc()

function to serve over HTTP the remote document $\mathcal{D}$, given the URL xrpc://host/$\mathcal{D}$. Another feature is *bulk* RPC, that allows to handle multiple calls to the same function (but with different parameters) in a single network interaction. Bulk RPC is exploited when a query contains a function call nested in an XQUERY for-loop, which in a naive implementation would lead to as many synchronous RPC network interactions as loop iterations. XRPC is implemented in *MonetDB/XQuery* [4], an open source XQUERY engine, which we use for experimental evaluation.

Figure 1 shows a query Q_e that performs a single XRPC function call to fcn(), with a single parameter (a node $n from some document $\mathcal{D}$). To make an XRPC call, the local peer formulates a SOAP request message (Simple Object Access Protocol is the XML-based message format used for webservices [11], [12], [19]), which contains a deep copy P of the node $n. That is, XRPC follows the previously mentioned approach of copying the XML subtree of a node parameter; this implies a *pass-by-value* parameter passing strategy. The message is sent as a synchronous HTTP POST request. The remote peer runs a HTTP server, which parses the request message and constructs a separate XML fragment for each node parameter (in this example a single fragment P'). The remote peer then evaluates the function and again serializes the result into a response message (here, a deep copy of the result node, denoted R). Finally, the local peer parses the response message and constructs a separate XML fragment for each node-typed result (here R'), which is the result of Q_e.

Problem Statement. Our goal is to rewrite an XQUERY Q that uses XML documents with xrpc:// URIs stored at remote peers, into an equivalent Q' that uses XRPC calls to execute parts of the query (expressed as XQUERY functions) on those remote peers. For a query Q, $Q(\mathcal{D})$ denotes the result of evaluating Q over a (possibly distributed) database $\mathcal{D}$. Two queries Q and Q' are *equivalent*, if $Q(\mathcal{D}) = Q'(\mathcal{D})$ for any given database $\mathcal{D}$ (the XQUERY *deep-equal* semantics).

We illustrate XQUERY decomposition as follows:

```
for $e in doc(“employees.xml”)//emp
where $e/@dept = doc(“xrpc://example.org/depts.xml”)//dept/@name
return $e
```

the URL xrpc://example.org/depts.xml implies that the remote peer example.org supports XRPC, to which the predicates could be pushed as:

```
declare function fcn($n as xs:string) as xs:boolean
{ $n = doc(“depts.xml”)//dept/@name };

for $e in doc(“employees.xml”)//emp
where execute at { “example.org” } { fcn($e/@dept) } return $e
```

In this example, the parameter and return value of the function fcn() are of atomic types. In more complex cases, nodes may be involved, such that potential semantic differences due to pass-by-value should be considered (discussed in Section II), which is our main challenge.

Contributions & Roadmap. Section II identifies the semantic differences of remote XQUERY pass-by-value function evaluation with respect to standard, local, function evaluation. Section III describes an XQUERY CORE based query decomposition framework. This leads in Section IV to a conservative XQUERY decomposition strategy that avoids semantic problems simply by refraining from decomposition in all problem cases. To make our rewrites more effective and robust against syntactic variation, we also describe normalization and code motion rewrite strategies. As a second contribution, Section V extends the pass-by-value semantics with a new *pass-by-fragment* message format, which conserves more structural relationships between nodes passed in a message, and allows more predicates to be distributed. Section VI then introduces a new *runtime* XML *projection* technique, which we use to generate messages that conserve all needed structural relationships between transferred XML nodes, and thus allow even more freedom in query decomposition. As a runtime technique, it is able to prune XML data much more than previously described compile-time projections [3], [5], [15]. An evaluation of the performance benefits of our techniques is given in Section VII. Finally, we discuss related work in Section VIII and conclude in Section IX with outlook on future work.

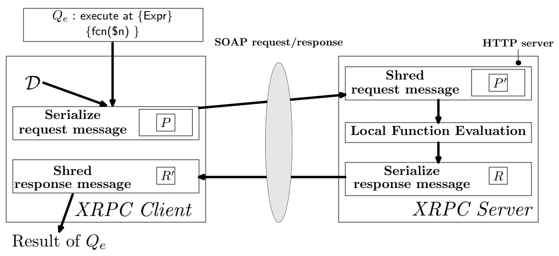


Fig. 1. XQUERY Remote Procedure Execution under Pass-by-value

```
 declare function makenodes() as node()
{⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩/b };                  ▷ node ⟨b⟩⟨c/⟩⟨/b⟩ has parent::a

declare function overlap($l as node(), $r as node()) as boolean
{ not(empty($l//∗ intersect $r//∗)) };             ▷ are $l and $r related?

declare function earlier($l as node(), $r as node()) as node()
{ if ($l≪$r) then $l else $r };

let $bc := makenodes(),
    $abc := $bc/parent::a                          ▷ $bc has a parent $abc
return  (for $node in ($bc, $abc)
           let $first := earlier($bc, $abc)                     ▷ always $abc
           where overlap($first, $node)                       ▷ always overlap
           return $node)//c                               ▷ returns only one ⟨c/⟩
```

TABLE I

EXAMPLE QUERY Q_1

II. SEMANTIC DIFFERENCES OF PASS-BY-VALUE

There are well-defined semantic differences [30] between evaluating an XQUERY expression locally and executing it remotely under pass-by-value parameter passing. We discuss these differences with a query Q_1 in Table I. This query evaluates three functions: makenodes(), overlap() and earlier().

Problem 1: Non-downward* XPATH *steps. Reverse and horizontal XPATH axis navigation (e.g., parent, ancestor, preceding(-sibling) and following(-sibling)) from remote function parameters always produces empty results, as pass-by-value node serialization only includes the descendants of a node inside the message. Consider the following:

```
let $bc := execute at {“example.org”} {makenodes()},
    $abc := $bc/parent::a
```

here, $abc evaluates to the empty sequence, instead of the correct a-node $\langle a\rangle\langle b\rangle\langle c/\rangle\langle /b\rangle\langle /a\rangle$.

It is possible to evaluate downward XPATH steps on a sequence of remote nodes, but only if we are sure that these nodes are ordered *and* non-overlapping (otherwise, the results of such XPATH steps will fail to respect node identity and order, as described below).

Problem 2: Node identity comparisons. If a remote function returns a sequence with two identical nodes, or two identical nodes are passed as function parameters, pass-by-value represents them as two different copies. This leads to the duplicate elimination problem described just above, and any *node identity* comparison will always yield false. For instance:

```
where execute at {“example.org”}{overlap($first, $node)}
```

yields false, while the local query evaluation gives true.

Problem 3: Document order. The parameters of a function call on a remote peer are serialized into the message in parameter order, in separate XML fragments. Even if the parameter nodes are disjoint (making *Problem 2* irrelevant), the relative order between these XML fragments may differ from their original order. Thus, inter-parameter node comparisons (“≪”, “≫”) may behave differently from the local semantics. Consider the usage of earlier() in Q_1 as:

```
let $first := execute at{“example.org”} {earlier($bc,$abc)}
```

In both iterations, the variable $first binds to a copy of $bc, instead of $abc, although $abc is the parent of $bc.

Problem 4: interaction between different calls. Additional semantic differences can occur when XQUERY subexpressions (sequences) may contain nodes that were obtained as results from *different* remote function calls, and these function calls, directly or indirectly, accessed the same XML document on some peer. Node sequences can become intermixed by any XQUERY construct that accepts multiple inputs, namely: sequence construction, and the built-in functions union, except, and intersect. A special source of call-mixing is the return clause of a for- loop in which remote function evaluation is performed, because the return clause implicitly creates a sequence that concatenates the expression result of all loop iterations (each of which performed a semantically separate remote function call). The result of such “mixed-call expressions” is that nodes returned by different calls may in fact stem from the same document; however node identity and ordering between nodes from different calls is not preserved, leading to semantic differences. For example, even if a downward XPATH step is applied on an input sequence containing nodes obtained from different remote calls, the result can have the wrong order (placing the results from the first call always before those of the second call) and will fail to properly eliminate duplicates:

```
(for $node in ($bc, $abc)
let $first := execute at {“example.org”}{earlier($node,$abc)}
return $node)//c
```

The above two XRPC calls produce nodes belonging to separate XML fragments. Under pass-by-value, evaluating //c produces two separate copies of c nodes, while in local execution the nodes returned from earlier() are from the same XML fragment, such that XPATH steps return a duplicate-free result.

```
1:  Expr         ::= ExprSingle | ExprSeq
2:  ExprSeq      ::= “()” | ExprSingle (“,” ExprSingle)+
3:  ExprSingle   ::= Literal | VarRef | ForExpr | LetExpr | IfExpr
                   | Typeswitch | CompExpr | OrderExpr | StepExpr
                   | NodeSetExpr | Constructor | FunCall
4:  VarRef       ::= “$”Var
5:  Var          ::= QName
6:  ForExpr      ::= “for” “$”Var “in” ExprSingle “return” ExprSingle
7:  LetExpr      ::= “let” “$”Var “:=” ExprSingle “return” ExprSingle
8:  IfExpr       ::= “if” “(” Expr “)” ThenElse
9:  ThenElse     ::= “then” ExprSingle “else” ExprSingle
10: Typeswitch   ::= “typeswitch” “(”Expr“)” CaseClause+
                   | “default” “$”Var “return” ExprSingle
11: CaseClause   ::= “case” “$”Var “as” SequenceType “return” ExprSingle
12: CompExpr     ::= ExprSingle (ValueComp | NodeCmp) ExprSingle
13: ValueComp    ::= “=” | “! =” | “<” | “<=” | “>” | “>=”
14: NodeCmp      ::= “is” | “≪” | “≫”
15: OrderExpr    ::= ExprSingle “order by” OrderSpecs
16: OrderSpecs   ::= ExprSingle (“ascending” | “descending”)(, OrderSpecs) ∗
17: NodeSetExpr  ::= Expr NodeSetOp Expr
18: NodeSetOp    ::= “union” | “intersect” | “except”
19: Constructor  ::= (“document” | “text”) “{” Expr “}”
                   | (“element”| “attribute”) (QName | “{”Expr“}”) “{”Expr“}”
20: StepExpr     ::= “/” | AxisStep “::” NodeTest
21: AxisStep     ::= RevAxis | FwdAxis | HorAxis
22: RevAxis      ::= “ancestor” | “ancestor-or-self” | “parent”
23: FwdAxis      ::= “self” | “child” | “attribute”
                   | “descendant” | “descendant-or-self”
24: HorAxis      ::= “preceding” | “preceding-sibling”
                   | “following” | “following-sibling”
25: NodeTest     ::= “node()” | “text()” | QName | “∗”
26: FunCall      ::= QName “(” (ExprSingle (“,” ExprSingle) ∗)? “)”
```

TABLE II

GRAMMAR OF EXTENDED XCORE RULES

Problem 5: XQUERY ***built-in functions.*** Various problems may occur when evaluating certain built-in functions remotely.

1) static-base-uri(), default-collation() and current-datetime(): depend on the *static* XQUERY context.
2) base-uri() and document-uri(): depend on the *dynamic* context of node expressions.
3) root(): accesses the document root.
4) id() and idref(): return all nodes in a document with certain id/idref values.

Class 1 of above built-in functions is handled by extending the XRPC message format with extra attributes such that the remote side can declare identical values for these context attributes. Class 2 is dealt with by adding these properties as attributes in the XRPC nodes (such as xrpc:element) that enclose serialized parameter/result nodes in the SOAP messages. Use of the fn:base-uri() and fn:document-uri() in XRPC is substituted by xrpc:base-uri() and xrpc:document-uri() wrappers that take these attributes into account when invoked on XRPC parameter nodes. As solutions for Class 1-2 are available, the main problem with built-in functions is posed by Classes 3-4, which return non-descendants of parameter nodes, and thus cannot be supported with pass-by-value.

In the remainder, we present decomposition techniques and extensions to enhance the pass-by-value semantics, that solve these problems.

III. XQUERY CORE REWRITE FRAMEWORK

XQUERY CORE [8] (abbreviated XCORE) is a subset of XQUERY, in which all implicit operations are made explicit.

We adopt a subset of XCORE expressions in Table II, which is sufficient to capture XPATH 1.0 and XQUERY FLWOR expressions [8]. We use a representation of XPATH paths in our XCORE grammar that keeps consecutive steps together, rather than nesting each step in a separate for-loop (when allowed – the use of position() precludes this). Such an optimization is common in XQUERY engines, and is part of XQUERY normalization, further described in Section IV. Additionally, we define two new rules for the XRPC extension [30]:

```
27: XRPCExpr   ::= “excute at” “{”ExprSingle“}” “function” XRPCParam “{”Expr“}”
28: XRPCParam  ::= “()” | “(” “$”Var “:=” VarRef (“,” XRPCParam)? “)”
```

Rule 27 identifies an xrpc:// URI in expression ExprSingle, and declares a new *anonymous function* that is to be executed remotely. It is noticeable that these grammar rules lack the expressive power to define recursive functions. This does not matter for XQUERY decomposition, as our decomposition strategies will not generate recursive functions. We also note that the syntax defined by rules 27 and 28 differs from the actual XRPC syntax (execute at {Expr}{FunApp(ParamList)}). The syntax used here is only for presentation purpose, to avoid the need to define all rules concerning declaration of user-defined functions. Thus, our simple XCORE rule without explicit user-defined function declarations allows to express all queries in a single Expr, which in turn can be mapped to a query graph. This simplifies the formulation of analysis steps.

A. XCore Dependency Graph

We introduce a dependency graph (d-graph) for an XCORE query. Consider the XQUERY query Q_2 in Table III, which asks for the grade in *course42* of students having a *tutor* who is also a *student*, and its XCORE equivalence Q_2^c.

A *dependency graph* is a directed, ordered and connected graph G with vertices $V(G)$ and edges $E(G)$. Each vertex v is denoted as v_i*:rule[val]*, where v_i is a unique vertex identifier, *rule* is the grammar rule represented by v_i, and *val* is an optional value indicating the right-hand-side of *rule*. There is a single *root* vertex without incoming edges. $E(G)$ consists of parse edges $E_p(G)$ and varref edges $E_v(G)$. Each *parse edge* is an ordered vertex pair (u, v), where u corresponds to a parsing rule r_u that directly causes the use of another parsing rule r_v. A *varref edge* is an ordered vertex pair (w, x) denoting a variable usage. When a VarRef rule is used, an additional edge is created between the VarRef vertex and the Var vertex that defines the variable.

Example 3.1: *Figure 2 shows the* d*-graph of* Q_2^c *in Table III. Solid and dashed lines represent parse and varref edges, respectively. For instance, the variable binding in the first* let *expression corresponds to vertices* $v_2, \ldots, v_7$*, and vertices* $v_8, \cdots, v_{39}$ *depict its* return *clause. The edge* (v_4, v_5) *is a parse edge. The edge* (v_{30}, v_9) *is a varref edge, as the variable used by* v_{30} *is a reference of variable* $\$c$ *introduced by* v_9*. Thus, a d-graph is in essence a parse-tree with additional (dashed) edges to indicate variable usages.*

We define three types of dependency relationships upon the reachability between two vertices x, y in $V(G)$: (1) x *“parse-depends on”* y, denoted as $x \overset{p}{\rightsquigarrow} y$, if y is reachable from x via *only* parse edges; (2) x *“varref-depends on”* y, denoted as $x \overset{v}{\rightsquigarrow} y$, if y is reachable from x via at least one varref edge; and (3) x *“depends on”* y, denoted as $x \rightsquigarrow y$, if either $x \overset{p}{\rightsquigarrow} y$ or $x \overset{v}{\rightsquigarrow} y$ holds. The compositional nature of XQUERY means that $x \rightsquigarrow y$ concisely captures all semantic dependencies between subexpressions.

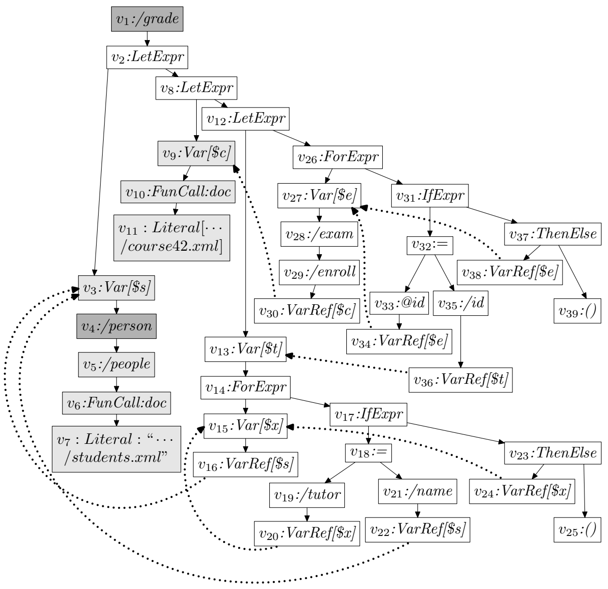


Fig. 2. D-graph

basic XQUERY **query**

```
(let $s := doc(“xrpc://A/students.xml”)/people/person,
     $c := doc(“xrpc://B/course42.xml”),
     $t := $s[tutor = $s/name]
 for $e in $c/enroll/exam
 where $e/@id = $t/id
 return $e)/grade
```
Q_2

XCORE **variant**

```
(let $s := doc(“xrpc://A/students.xml”)/child::people/child::person return
   let $c := doc(“xrpc://B/course42.xml”) return
     let $t := for $x in $s return
                     if ($x/child::tutor = $s/child::name) then $x else ()
     return for $e in $c/child::enroll/child::exam return
                  if ($e/attribute::id = $t/child::id) then $e else ())/child::grade
```
Q_2^c

normalized XCORE **variant**

```
(let $t :=  ( let $s := doc(“xrpc://A/students.xml”)/child::people/child::person
                return for $x in $s return
                           if ($x/child::tutor = $s/child::name) then $x else () )
 return for $e in (let $c := doc(“xrpc://B/course42.xml”
                         return $c/child::enroll/child::exam)
        return if ($e/attribute::id = $t/child::id) then $e else ())/child::grade
```
Q_2^n

TABLE III
EXAMPLE QUERY Q_2

Consider Figure 2, $v_{15} \overset{p}{\rightsquigarrow} v_{16}$, since (v_{15}, v_{16}) is a parse edge; $v_{15} \overset{v}{\rightsquigarrow} v_3$, as v_3 is reachable from v_{15} via $(v_{15}, v_{16}), (v_{16}, v_3)$ and (v_{16}, v_3) is a varref edge.

For a d-graph G and a vertex $r_s \in V(G)$, we use the term *subgraph* to mean the vertex-induced subgraph of r_s, including r_s and all $u \in V(G)$ where $r_s \overset{p}{\rightsquigarrow} u$; r_s is called the *root* of the subgraph. For instance, the subgraph rooted at vertex v_{26} contains vertices $v_{26}, \cdots v_{39}$, but does not contain vertices $v_{13}, \ldots, v_{25}$. Throughout this paper, we use the terms (sub)graph and (sub)query interchangeably, as a (sub)query is represented by the induced subgraph rooted at some vertex.

B. XRPCExpr Insertion

We can decide to evaluate a certain subgraph G_s rooted at r_s remotely over XRPC, by inserting a v_x:XRPCExpr node above it. This should only be done if we can ensure that the result of the rewritten query is identical to the original query. Such an insertion means that a new function will be defined that contains G_s as its body. In the main query graph, G_s is replaced by a remote XRPC call to this function, which receives as parameters all variable references in G_s that resolve to variable bindings outside G_s:

1) Insert a vertex v_x:XRPCExpr, a parse edge (v_x, r_s), and replace each incoming edge (v_{in}, r_s) with a new edge (v_{in}, v_x).
2) For each outgoing varref edge from vertex $v_i \in V(G_s)$ to $v_j \in V(G) \backslash V(G_s)$, where edge $(v_i, v_j) \in E_v(G)$ is a varref edge as (v_i:VarRef[$\$qname$], v_j:Var[$\$qname$]), we insert a new vertex v_k, a new parse edge (v_x, v_k) and replace the varref edge (v_i, v_j) by (v_i, v_k) and (v_k, v_j). Here, v_k has the form v_k:XRPCParam[$\$p$:=$\$qname$], which introduces a new variable $\$p$ and binds it to $\$qname$ in v_j.
3) If there are no outgoing edges as stated in *step* 2, we insert a vertex v_l with the form v_l:XRPCParam[()] (i.e., empty parameter), and a parse edge (v_x, v_l).

Example 3.2: *Consider the d-graph in Figure 2. Suppose that the subgraph rooted at v_{26} is identified for an* XRPCExpr *insertion (Figure 3). First, insert vertex v_{40} and replace edge (v_{12}, v_{26}) by (v_{12}, v_{40}) and (v_{40}, v_{26}). For the outgoing varref edge (v_{30}, v_9), vertex v_{41} is inserted below v_{40} and the varref edge is replaced by two new varref edges: (v_{30}, v_{41}), (v_{41}, v_9). The outgoing varref edge (v_{36}, v_{13}) is processed similarly.*

IV. Conservative Decomposition

By-value insertion conditions. Given a d-graph G and a subgraph G_s of G rooted at r_s, under the pass-by-value semantics, vertex r_s is in the set $I(G)$ of *valid decomposition points* (d-points), iff r_s satisfies all of the following conditions:

i. $(\nexists n \in V(G) : n.rule \in \{\text{RevAxis}, \text{HorAxis}\}) \wedge (\text{useResult}(n, r_s) \vee \text{useParam}(n, r_s))$;

ii. $(\nexists n \in V(G) : n.rule \in \{\text{NodeCmp, NodeSetExpr}\}) \wedge (\text{useResult}(n, r_s) \vee \text{useParam}(n, r_s))$;

iii. $(\nexists n \in V(G) : n.rule = \text{AxisStep} \wedge \exists m \in V(G) : m.rule \in \{\text{ForExpr, OrderExpr, ExprSeq, NodeSet-Expr, AxisStep} \backslash \{\text{parent, self, child, attribute}\}\} \wedge ((\text{useResult}(n, r_s) \wedge r_s \rightsquigarrow m) \vee (\exists v \in V(G) \backslash V(G_s) : r_s \overset{p}{\rightsquigarrow} n \rightsquigarrow v \rightsquigarrow m))$;

iv. $\nexists n \in V(G)$: $n.rule$=FunCall $\wedge$ $n.val$ $\in$\{fn:root(), fn:id(), fn:idref()\} $\wedge$(useResult$(n, r_s)\vee$ useParam(n, r_s)).

where we impose these restrictions symmetrically both on expressions that use the result of the remote expression r_s, as well as on the way remote expressions (below r_s) use their shipped parameters:

$$\text{useResult}(n, r_s) \Leftrightarrow n \rightsquigarrow r_s$$
$$\text{useParam}(n, r_s) \Leftrightarrow \exists v \in V(G) \backslash V(G_s) : r_s \overset{p}{\rightsquigarrow} n \rightsquigarrow v$$

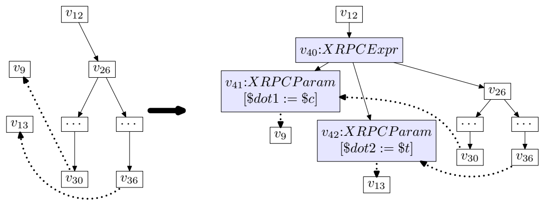


Fig. 3. XRPCExpr Insertion

Conditions i, ii guard against using any node comparision as well as horizontal and reverse XPATH steps on shipped nodes, avoiding *Problems* 1, 2, and 3 described in Section II. Condition iii avoids using (downwards, per Condition i) XPATH steps on shipped nodes stemming from expressions that might be so-called "mixed-call sequences" (ExprSeq, NodeSet-Expr), avoiding *Problem* 4. It also guards against sequences not in node order (ForExpr, OrderExpr) or with nodes that may be overlapping (the restrictions on XPATH steps). This ensures that downwards XPATH steps can be used on shipped node sequences. Condition iv states that the subgraph G_s may not contain an application of the listed built-in functions (*Problem* 5).

Example 4.1: *In the d-graph of example query Q_2^c (Figure 2), we mark in gray colors the d-points identified by the conservative decomposition strategy. The* XPATH *step* /grade *that is performed on the result of a* for*-loop, matches condition iii and causes all vertices that depend on v_{14} and v_{26} (the* For Exps*) as well as all their descendants to be excluded from $I(G)$, leaving v_1 and the subgraphs rooted at v_3, v_9 as d-points.*

Interesting decomposition points. While a d-point may be semantically valid, remote evaluation of the subquery below it might be senseless. Consider the d-point v_9, which contains only a fn:doc() function call in its subgraph. Executing this function remotely provides no performance gain, as it only demands the shipping of a whole document. Similarly, remote execution of expressions that do not involve any XML documents should be avoided. Therefore, we filter d-points by first annotating each vertex $v_x \in V(G)$ with the URI *dependency set* $D(v_x)$. Here, $D(v_x)$ represents the set of URIs that are used as parameters of fn:doc() in vertices that the vertex v_x can reach via parse edges:

$$D(v_x) = \{uri::v_y | \{v_y,v_z\} \in E(G) : uri = v_z.val \wedge v_x \overset{p}{\rightsquigarrow} v_y$$
$$\wedge v_y.rule = FunApp \wedge v_y.val = \text{"}doc\text{"} \wedge v_z.rule = Literal\}$$

Note that we tag each uri with the vertex v_y where the document is opened, to be able to distinguish the use of the same document through multiple fn:doc() calls. This definition does not cover the case that the parameter of fn:doc() is an expression instead of a literal. In those cases, we use a wildcard symbol "$*$" as uri. In this paper, the built-in function fn:collection() is treated as a fn:doc(*), and an element construction is assigned an artificial unique URI fn:doc(v_i::v_i).

One can use the URI dependency set to partition the $V(G)$ in *equivalence classes* (i.e., those vertices with the same URI dependency set belong to the same class). Using all vertices in an equivalence class, we can consider its induced subgraph in G, and try to handle it in a single XRPC subquery. Thus, we define *interesting decomposition points* (i-points) $I'(G)$ as those valid insertion points that *(a)* are a root vertex in their induced subgraph[1] *(b)* contain at least one fn:doc() and *(c)* execute at least one XPATH step on the fn:doc() function:

$$\begin{aligned} I'(G) = \{v_x | v_x \in I(G) : & \nexists v_y : v_y \overset{p}{\rightsquigarrow} v_x \wedge D(v_x) = D(v_y) \\ & \wedge \exists v_z : v_x \overset{p}{\rightsquigarrow} v_z \wedge v_z.rule = AxisStep \\ & \wedge \exists \text{xrpc://}uri \in D(v_x)\} \end{aligned}$$

Example 4.2: *In Figure 2, the two subtrees rooted at v_3 and v_9, that together form the d-points corresponding to two different equivalent classes $D(v_3) = \{$*xrpc://A/students.xml :: *$v_7\}$ and $D(v_9) = \{$*xrpc://B/course42.xml :: *$v_{11}\}$. Since the subtree of v_9 lacks an* XPATH *step, it does not have any i-points per restriction (c). The vertices in $I'(G)$ (colored dark gray) are v_4 (the highest non* Var *vertex in the subtree rooted at v_3) and the root v_1. Thus, $I'(G) = \{v_1, v_4\}$.*

Normalization. Rewriting algorithms that operate on the XCORE level are vulnerable to syntactic variation. In the case of our decomposition strategy, an important vulnerability comes from the behavior of the strategy to ship with XRPC subgraphs consisting of parse-edges only. That is, varref-edges are not pushed, rather become parameters to the function. The syntactic freedom one has in XQUERY of defining a certain subexpression inline or via a variable reference to a previous let-binding, therefore affects our strategy. For this purpose, as part of XCORE normalization, we re-order let-bindings, moving them as deep into the query as possible. More specifically, let-bindings are moved to just above the lowest common ancestor vertex (defined in terms of parse-edges) of all vertices that reference its variable. The query Q_2^c (Table III) can be normalized to Q_2^n (Table III), which can thus be rewritten as Q_2^v in Table IV.

The main achievement of normalization in above case is to relate the doc(../course42.xml) call through parse-edges (directly calling $\$c$ in Q_2^n), instead of varref edges (referencing $\$c$ in Q_2^c), with its use in the /child::enroll/child::exam XPATH steps. However, these being part of a ForExpr with the /grade step on top, causes insertion condition ii to prohibit pushing it. In the next section on pass-by-fragment, however, we will see that normalization was not in vain, and the query can be decomposed into Q_2^f (Table IV).

Distributed code motion. The let-normalization phase has the effect of pushing expressions that depend on the same documents downwards, potentially below an interesting insertion point (which makes them be executed remotely). However, it can happen that some of the expressions initially found below an interesting insertion point can in fact better be moved above it (to be executed locally). In particular, it is safe to assume that expressions that solely depend on a parameter of a function, can better be evaluated on the caller side. Moving a subexpression out of a function can be done by passing that subexpression as an additional parameter to the function. With pass-by-value passing, such a rewrite may

[1]If the root node happens to be a Var vertex, we consider its value expression instead as root.

Q_2^v: decomposed Q_2^n under pass-by-value

```
declare function fcn1() as node()*
{ doc("xrpc://A/students.xml")/child::people/child::person };

declare function fcn0() as node()*
{ (let $t := let $s := execute at{'A'}{fcn1()}
                  return for $x in $s return
                          if ($x/child::tutor = $s/child::name) then $x else ()
   return for $e in (let $c := doc("xrpc://B/course42.xml") return
                              $c/child::enroll/child::exam)
      return if ($e/attribute::id = $t/child::id) then $e else () )/child::grade };

execute at {} {fcn0()}
```

Q_2^f: decomposed Q_2^n under pass-by-fragment

```
declare function fcn1() as node()*
{ let $s := doc("xrpc://A/students.xml")/child::people/child::person return
      for $x in $s return if ($x/child::tutor = $s/child::name) then $x else () };

declare function fcn2($para1 as node()) as node()*
{ for $e in (let $c := doc("xrpc://B/course42.xml") return
                        $c/child::enroll/child::exam)
   return if ($e/attribute::id = $para1/child::id) then $e else () };

declare function fcn0() as node()*
{ let $t := execute at {'A'}{fcn1()} return
     (execute at {'B'}{fcn2($t)})/child::grade };

execute at {} {fcn0()}
```

Applying Distributed Code Motion in Q_2^f

```
declare function fcn2new($para2 as xs::string∗) as node()*
{ for $e in (let ··· return ···) return
      if ($e/attribute::id = $para2) then $e else () };

declare function fcn0() as node()*
{ let $t := execute at {'A'}{fcn1()} return
    let $l := $t return (execute at {'B'}{fcn2new($l/child::id)})/child::grade };
```

TABLE IV

QUERY DECOMPOSITION AND CODE MOTION

not always be safe, however if only d-points are moved, the technique is semantically safe. Analogous to well-known compiler technique of moving invariant statements out of the loop (and its use in parallel processing [14]) we call this technique *distributed code motion*.

Example 4.3: *Consider the function* fcn2()*, we may observe that the expression* $para1/child::id *only depends on the function parameter* $para1*. Shipping full* person *nodes* $para1 *from peer* A *to* B*, only to extract the string value of its* id *child at* B*, may waste bandwidth, especially if* person *carries much more data than just an* id*. Instead, it would be better to extract the string value of* id *at peer A and only ship the strings. This optimization can be realized by adding a new parameter* $para2 *to the function, and substituting* $para1/child::id *in the body with it. In the function* fcn0() *that calls* fcn2new()*, we save the original function parameter* $t *in a new* let*-binding* $l*, and pass* $l *instead of* $t*. The additional function parameter is passed as* $l/child::id*. Finally, the affected function parameter* $para1 *is no longer used, so we remove it, arriving at the result as the code motion part in Table IV.*

V. BY-FRAGMENT DECOMPOSITION

The node copying done by pass-by-value is the main source of semantic differences. This, in turn, leads to serious restrictions in the way the decomposition strategy can push expressions remotely. For this reason, we extend the pass-by-value message passing semantics into a new *pass-by-fragment* message passing semantics that better preserves structural XML relationships.

The basic idea is to avoid serializing the same nodes twice, by grouping all node-valued data in the message in a preamble element fragments. In principle, each node parameter is serialized below a separate fragment child element. However, if a sent node is a descendant of another one, it is not serialized twice, as we can reuse the XML fragment of the other node. We also ensure that the XML fragments are sorted in original document order, which means that ancestor/descendant relationships in the same message, as well as node identity and document order, are preserved.

Later in the message, where XQUERY sequences are serialized (inside sequence tags), we just provide references to the nodes that were previously serialized in the fragments. In particular, element tags, which are used to contain as a child the fully serialized copy of a node, now just carry two numeric attributes, fragid and nodeid. Supposing \$msg is the root of the message, with \$fragid and \$nodeid numbers, we can identify the referenced nodes as follows:[2]

```
$msg//fragment[$fragid]/descendant::node()[$nodeid]
```

Example 5.1: *Going back to Q_1 in Table I, Figure 4 shows the* XRPC *request message sent for the call* execute at {"example.org"} {earlier (\$bc, \$abc)} *from the discussion of Problem* 3. *Recall that the node* \$bc *with value $\langle b\rangle\langle c/\rangle\langle /b\rangle$ is contained in the* \$abc *fragment $\langle a\rangle\langle b\rangle\langle c/\rangle\langle /b\rangle\langle /a\rangle$. The lower part of the figure shows an excerpt of the message as produced for pass-by-fragment. Here, both node parameters* \$bc *and* \$abc *are represented in* element *nodes with* fragid *and* nodeid *attributes. The* XQUERY *engine handling the call will use these attributes to evaluate:*

```
$bc := $msg:fragment[1]/descendant::node()[2],
$abc := $msg:fragment[1]/descendant::node()[1]
```

such that earlier(\$bc,\$abc) *correctly returns* \$abc, *because* \$abc $\ll$ \$bc, *just like on the peer that invoked this function. The upper part, with the changed part of the old pass-by-value message (element* call*), shows that node parameters were previously repeatedly serialized, causing node order and identity relationships between parameters to be lost.*

We made a conscious choice not to rely on ID/IDREF for referencing nodes, since this would require adding ID attributes to the XML data in the fragments. As XRPC is designed to respect and conserve XML SCHEMA type information, this would cause the XRPC message to no longer respect user-defined schemas.

By-fragment insertion conditions. With the pass-by-fragment semantics, we modify the pass-by-value decomposition conditions listed in Section IV by restricting the prohibitions to decompose a node r_s formulated in Conditions ii and iii to only those r_s for which the predicate hasMatchingDoc(r_s) holds. Here, hasMatchingDoc() is defined as:

$$\text{hasMatchingDoc}(v) \Leftrightarrow \forall uri_l{::}v_i \in D(v) : \exists uri_r{::}v_j \in D(v) : v_i \neq v_j \wedge (uri_l = uri_r \vee uri_l = * \vee uri_r = *)$$

[2]Note that descendant::node() does not return attribute nodes. We use the nodeid of its parent and include the name of the attribute in an attribute element, so it can be found back with an additional attribute step.

Excerpt of a request message with pass-by-value

```
⟨call⟩
  ⟨sequence⟩⟨element⟩⟨b⟩⟨c⟩⟨/b⟩⟨/element⟩⟨/sequence⟩
  ⟨sequence⟩⟨element⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/element⟩⟨/sequence⟩
⟨/call⟩
```

Excerpt of pass-by-fragment message for earlier(\$bc,\$abc)

```
⟨env:Envelope ...⟩
  ⟨env:Body⟩
    ⟨request⟩
      ⟨fragments⟩
        ⟨fragment⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/fragment⟩
      ⟨/fragments⟩
      ⟨call⟩
        ⟨sequence⟩⟨element fragid="1" nodeid="2"⟩⟨/sequence⟩
        ⟨sequence⟩⟨element fragid="1" nodeid="1"⟩⟨/sequence⟩
      ⟨/call⟩
    ⟨/request⟩
  ⟨/env:Body⟩
⟨/env:Envelope⟩
```

Fig. 4. By-value vs. By-fragment Messages

This predicate now precisely isolates the problem of creating result sequences with remote nodes from multiple calls, by stating that an expression may not depend on two *different* applications in the query of fn:doc() with the *same* URI (taking into account computed URIs as wildcards). Additionally, we also remove from Condition iii, the ForExpr restriction (a special form of combining the results of multiple calls), since Bulk RPC ensures that all iterations of the remote call nested in the `for`-loop, are handled in a single message exchange (where copy-by-fragment now ensures proper conservation of node relationships). Finally, we remove from Condition iii the restriction that all pushed steps should be of the non-overlapping kind (parent, self, child, attribute), as the copy-by-fragment message passing is able to properly conserve the ancestor/descendant relationships between nodes. The restrictions to avoid horizontal and reverse XPATH steps on remote nodes (Condition i) and on using built-in functions (Condition iv) remain in place here; however we will be able to remove both fully in the sequel.

VI. By-Projection Decomposition

The basic idea of using XML projection [18] is, for a given XQUERY query Q and an XML document $\mathcal{D}$, to extract a minimal part of data $\mathcal{D}'$ needed to execute Q such that $Q(\mathcal{D}) = Q(\mathcal{D}')$. The projection technique conducts a compile-time path analysis on Q, to derive a set of simple path expressions that over-estimate the nodes that Q touches. These simple paths are referred to as *projection paths*. Here, a *projection path* is an XML path that starts from the document root, containing forward navigation but not predicates (e.g., $doc(\$uri)/a/b/@id$). Projection paths consist of returned paths and used paths. *Returned paths* describe the nodes that are returned by the expression. *Used paths* estimate the nodes necessary to answer the query but are never returned as results (e.g., predicates).

Based on the projected paths $\mathcal{P}$ of query Q from path analysis, a loading algorithm is applied to $\mathcal{P}$ and an XML document (from a file or a stream) $\mathcal{D}$. A projected XML document (or stream) $\mathcal{D}'$ is then generated which contains all used and returned nodes plus the descendants of the returned nodes, and is queried with Q.

Excerpt of request message for makenodes()

```
⟨request⟩
  ⟨projection-paths⟩
    ⟨used-path/⟩
    ⟨returned-path⟩parent::a⟨/xrpc:returned-path⟩
  ⟨/projection-paths⟩
  ⟨fragments/⟩
```

Excerpt of response message for makenodes()

```
⟨env:Envelope ...⟩
 ⟨env:Body⟩
  ⟨response⟩
   ⟨fragments⟩
    ⟨fragment⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/fragment⟩
   ⟨/fragments⟩
   ⟨call⟩
    ⟨sequence⟩⟨element fragid="1" nodeid="2"⟩⟨/sequence⟩
   ⟨/call⟩
  ⟨/request⟩
 ⟨/env:Body⟩
⟨/env:Envelope⟩
```

Fig. 5. Pass-By-Projection Messages

There are three reasons why projecting XML is extremely interesting for distributed XML processing: *(i)* until now, when sending nodes, we had to serialize all descendants – which potentially contain huge subtrees that may remain untouched on the other side. This amounts to wasted network bandwidth as well as serialization and shredding effort. *(ii)* if documents are projected into lean skeletons that only contain the relevant portions, it becomes feasible to serialize XML fragments from some *lowest common ancestor* on, possibly even the document root. Even with pass-by-fragment, the execution of reverse/horizontal XPATH axes on remote nodes is impossible. By extending projecting XML with support for reverse and horizontal axes, however, we get a tool to precisely identify this lowest common ancestor of an XML document that needs to be included to allow correct remote execution of those axes. *(iii)* the projection technique can even be applied to support the built-in functions fn:root() and fn:id()/fn:idref(), i.e., by taking the lowest common ancestor of those, if a path contains one of these functions.

For these reasons, we further refine the pass-by-fragment message passing semantics into the so-called *pass-by-projection* semantics. XML projection can be used in both directions: to project the parameters in a request message, and to project the function's result sequence before shipping back the response.

Insertion conditions. Pass-by-projection removes the by-fragment insertion conditions (in Section V) i and iv, such that only ii and iii, i.e., the application of node comparison, node set operators and axis steps on top of multiple calls to fn:doc() with the same URI, remains illegal.

Message extension: projection paths. We introduce an optional element as a sub-element of a request tag: projection-paths, which in turn has zero or more child elements returned-path and used-path. In the new pass-by-projection semantics, the absence or presence of this element determines whether the response message should be in the original pass-by-value or the new pass-by-projection format.

Example 6.1: *To illustrate projected XRPC messages, the upper part of Figure 5 shows part of the request message for the call from Q_1 (discussed in Problem 4):*

let \$bc := execute at {"example.org"} {makenodes()}

since the projection path analysis detects that \$bc *will subsequently be used as context node by a parent step:* \$abc := \$bc/parent::a, *the request message specifies* parent::a *as a returned path. Therefore, the response message contains the full fragment* $\langle a\rangle\langle b\rangle\langle c/\rangle\langle /b\rangle\langle /a\rangle$ *to which* \$abc *then gets correctly bound.*

$ProjectionPath$	::=	**doc** "(" Literal "::" Literal ")" (*"/" SimplePath*)*
$SimplePath$	::=	$Axis\ NodeTest$ | $SimplePath$ *"/"* $Axis\ NodeTest$
$Axis$	::=	"self::" | "child::" | "attribute::"
	|	"descendant::" | "descendant-or-self::"
	|	"**ancestor**::" | "**ancestor-or-self**::" | "**parent**::"
	|	"**root()**" | "**id()**" | "**idref()**"
$NodeTest$	::=	$((NCName\|*)$:)?$(NCName\|*)$| "node()" | "text()"

TABLE V

GRAMMAR RULE EXTENSION OF *ProjectionPath* (BOLD)

A. Extending Projected XML

We extend the path grammar rules [18] and path annotations, to handle full-fledged XQUERY involving reverse/horizontal XPATH steps and built-in functions. The extended grammar rule for *ProjectionPath* is given in Table V.

We denote path annotations in projected XML as follows:

$$Env(v_i) \vdash Expr \Rightarrow Paths_1 \text{ using } Paths_2$$

The notation $Env(v_i)$ is used to identify the path annotation environment at a certain vertex v_i in the XQUERY d-graph.

Such annotations are constructed bottom up by *path analysis rules* that derive the used ($UPaths$) and returned ($Paths$) paths for each XCORE expression in terms of used and returned paths of its subexpressions. The basic path analysis rules have been discussed in [18], such as *literal values*, *sequences*, for and let expressions and XPATH steps, etc. Our extension to include reverse/horizontal XPATH steps brings no changes for the path analysis rules, but must be supported by the loading algorithm, which is described in Section VI-B. We complement the rules for built-in functions, which apart from the unsolved cases mentioned under *Problem* 5 in Section II (fn:root(), fn:id(), fn:idref()) also includes fn:doc(). The description of the basic projection technique assumes a single document. As in distributed query processing there are always multiple documents, our paths always start with fn:doc(URI).

Path analysis rules. We provide one rule for fn:doc() with a constant parameter and another for computed URIs:

$$(\text{DOC}_1)\quad \frac{()}{Env(v_i)\vdash \text{doc}(Literal_1)\Rightarrow \text{doc}(Literal_1{::}v_i)\ \text{ using } \emptyset}$$

$$(\text{DOC}_2)\quad \frac{Env(v_j)\vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j}{Env(v_i)\vdash \text{doc}(Expr_j)\Rightarrow \text{doc}(*{::}v_i)\ \text{using} Paths_j \cup UPaths_j}$$

As mentioned in Section IV, in the definition of $D(v_x)^3$, we use a wildcard URI $*$ if the document name is an expression. Also note that all paths start with doc(URI::v_i), thus identifying

[3] We use the doc(..) prefixes of the *returned paths* annotations on v as a more precise form of the $D(v)$ property. Documents that were only used but not returned were also be part of the original $D(v)$, but these will not cause semantic problems.

both document URI as well as the vertex v_i where it is loaded. This notation facilitates the identification of situations where the same URI is loaded twice (the function hasMatchingDoc()). A similar rule can be formulated for XML element construction, producing a return path doc(v_i::v_i) with an artificial unique URI. The rule for fn:root() is:

$$\text{(ROOT)}\quad \frac{Env(v_j) \vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j}{Env(v_i) \vdash fn\!:\!root(Expr_j) \Rightarrow \cup_{p \in Paths_j} p\text{/root() using } UPaths_j}$$

The built-in function fn:root() with a single parameter is treated in the path annotations much like XPATH axis steps, where the parameter has become the path prefix. In this path notation, functions remain easily recognizable by the parentheses. The rules for the built-in functions fn:id()/fn:idref(), are highly similar (only fn:id() provided):

$$\text{(ID)}\quad \frac{\begin{array}{r} Env(v_j) \vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j \\ Env(v_k) \vdash Expr_k \Rightarrow Paths_k \text{ using } UPaths_k \end{array}}{\begin{array}{r} Env(v_i) \vdash fn\!:\!id(Expr_j, Expr_k) \Rightarrow \cup_{p \in Paths_k} p\text{/id()} \\ \text{using } Paths_j \cup UPaths_j \cup UPaths_k \end{array}}$$

The first parameter of fn:id() is ignored by the annotations, as it contains string values, and the projection annotation framework only allows for the estimation of node sets. This has the consequence that our loading algorithm will conserve *all* elements with an ID/IDREF attribute.

B. Runtime XML Projection

The extensions we made to XML projection, namely support for reverse/horizontal XPATH axes and fn:root(), fn:id()/fn:idref(), could not be trivially integrated in the loading algorithm of [18]. However, in this case we are not really looking for a loading algorithm that efficiently reads (shreds) an XML file into a projected representation. Rather, the documents are already present (and indexed) in the XQUERY engine, and runtime message projection is a *serialization* task. Therefore, we propose a new *runtime* approach for projection, targeted at serialization, rather than at shredding. Whereas the original loading algorithm starts at the document root, and evaluates *absolute* used and returned paths, our runtime projection algorithm starts in a run-time state, that is, with a real, *materialized context sequence* (e.g., the parameter values that are about to be serialized in a SOAP message), and executes only *relative* paths on them. Because the node sequence bound at runtime to a function parameter is only a subset of the node set characterized by its compile-time path annotation (e.g., its contents may well have been reduced by applying a selection predicate), this runtime projection technique can be much more precise than the original projection algorithm.

For these reasons, our *runtime* approach for projection simply relies on the normal XPATH evaluation capabilities of the XQUERY engine for fully evaluating all used and returned path annotations one-by-one (and uniting them with union()). Doing so, it produces a *used node set U* and a *returned node set R*. These two sets are the input for the runtime projection algorithm listed in Algorithm 1.

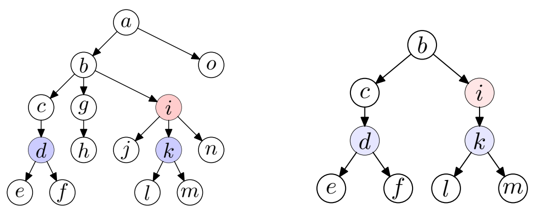
Algorithm 1: RUNTIMEXMLPROJECTION($U, R, \mathcal{D}$)

```
input : $U$- used nodes, sorted on document order
        $R$- returned nodes, sorted on document order
        $\mathcal{D}$- the original XML document
output: $\mathcal{D}'$- the projection of $U$ and $R$ on $\mathcal{D}$
1  projection nodes $P \leftarrow U \cup R$ and sorted on document order;
2  $proj \leftarrow$ first node in $P$;
3  $cur \leftarrow$ first node of $\mathcal{D}$, i.e., root node;
4  while $\neg P.end()$ do
5      if $proj$ is a descendant of $cur$ then
6          add $cur$ to $\mathcal{D}'$;
7          $cur \leftarrow$ next node in $\mathcal{D}$;
8      else if $proj = cur$ then
9          if $proj$ is a returned node then
10             add $cur$ and all descendants of $cur$ to $\mathcal{D}'$;
11             $cur \leftarrow$ next following node of $cur$ in $\mathcal{D}$;
12             while $proj.next$ is a descendant of $proj$ do
13                 $proj \leftarrow proj.next$          ▷ prune projection nodes;
14             end
15         else
16             add $cur$ to $\mathcal{D}'$;
17             $cur \leftarrow$ next node in $\mathcal{D}$;
18         end
19         $proj \leftarrow proj.next$                   ▷ next projection node;
20     else
21         $cur \leftarrow$ next following node of $cur$ in $\mathcal{D}$;
22     end
23 end
24 $cur \leftarrow$ root node of $\mathcal{D}'$;
25 while $cur$ has only one child node $\wedge\, cur \notin \{U \cup R\}$ do
26     $cur \leftarrow$ first child of $cur$;
27 end
```

Projection algorithm identifies all projection nodes in the XML tree representation of the original document, by traversing the tree top-down depth-first. During traversal, if the current node cur of the XML document is an ancestor of the current projection node $proj$ (line 5), cur is added to output $\mathcal{D}'$ and moved to the *next node* in document order. If a $proj$ is found (line 8), $proj$ is added to $\mathcal{D}'$; if this $proj$ is a returned node, all its descendants are also appended. Then cur is moved to its *next following node* in the document. Otherwise, if the current projection node $proj$ is not a descendant of cur, the subtree of cur can be skipped (line 21). Though this algorithm is formulated on an abstract level that is independent of the particular XML storage scheme used in an XQUERY engine, it is safe to assume that skipping a subtree is fast (either $O(1)$ or $O(log(|\mathcal{D}|))$). At the end of the algorithm (lines 24-27), post-processing is performed to remove unnecessary nodes, as we are only interested in the *lowest common ancestor* of all input nodes in the projected document $\mathcal{D}'$.

Example 6.2: *Consider an XML document $\mathcal{D}$ in Figure 6(a). Assume that the used node set U is $\{i\}$, and the returned node set R is $\{d, k\}$. Figure 6(b) shows the projected document $\mathcal{D}'$ of applying Algorithm 1 on U, R and $\mathcal{D}$.*

The algorithm starts with $P \leftarrow \{d, i, k\}$, $proj \leftarrow d$ and $cur \leftarrow a$. We traverse the tree using cur from a to d. Nodes a, b and c are added to $\mathcal{D}'$, since they are ancestors of the current context node d. Nodes d, e and f are also added to $\mathcal{D}'$, as d is a returned node. Then, cur is advanced to g (d's next following node). Because the next context node i is not in



(a) Original XML tree $\mathcal{D}$ (b) Projected tree $\mathcal{D}'$

Fig. 6. Runtime XML Projection Example

the subtree of g, the subtree is skipped by advancing cur to i. Recall that i is a used node, thus only i is added to $\mathcal{D}'$. The last context node is k. Our current document node cur traverses from i to j, and then to k, where we can add nodes k, l and m to $\mathcal{D}'$. The traversal can be terminated, because there is no more context nodes to process. However, the intermediate result $\mathcal{D}'$ contains all *common ancestors of $\{d, i, k\}$. The post-processing removes node a from $\mathcal{D}'$, which produces the final projected document $\mathcal{D}'$ as shown in Figure 6(b).*

Relative projection paths. At *compile time*, the XQUERY compiler builds a query graph (d-graph) with root v_{root}, normalizes it followed by decomposition and code motion. For each inserted XRPCExpr v_{xrpc}, and for each XRPCParam parameter vertex v_{param}, it then extracts the relative paths:

$U_{rel}(v_{param})$ = allSuffixes$(R(v_{param}), U(v_{xrpc}))$
$R_{rel}(v_{param})$ = allSuffixes$(R(v_{param}), R(v_{xrpc}))$
$U_{rel}(v_{xrpc})$ = allSuffixes$(R(v_{xrpc}), U(v_{root}))$
$R_{rel}(v_{xrpc})$ = allSuffixes$(R(v_{xrpc}), R(v_{root}))$, with:
allSuffixes$(Paths_i, Paths_j)$=$\{s_j | p_i/s_j \in Paths_j : \exists p_i \in Paths_i\}$

At *runtime*, $\cup_{\forall v_{param}} U_{rel}(v_{param})$ and $\cup_{\forall v_{param}} R_{rel}(v_{param})$ are used to project the parameters in the outgoing XRPC request message. The $U_{rel}(v_{xrpc})$ and $R_{rel}(v_{xrpc})$ are passed in the projection-paths element such that the remote peer can appropriately apply these paths to project the response message.

Projecting a document with Algorithm 1 requires pre-calculated used and returned node sets. These sets are simply computed using the XPATH evaluation infrastructure of the underlying XQUERY engine, by feeding the intermediate result \$ctx$_{param}$ corresponding to v_{param} as context sequence into all suffix paths $s_i \in U_{rel}(v_{param})$ (resp. $R_{rel}(v_{param})$):

union(\$ctx$_{param}/s_1$, union(\$ctx$_{param}/s_2$,....
union(\$ctx$_{param}/s_{n-1}$, \$ctx$_{param}/s_n$)...))

Paths \$ctx$/path_i/$root()$/path_j$ with function fn:root() are executed as root(\$ctx)$/path_j$. Similarly, \$ctx$/path_i/$id()$/path_j$ is executed as root(\$ctx)//attribute()::$(a_1|..|a_n)/../path_j$, where $a_1, .., a_n$ are all ID attributes (resp. IDREF in case of idref()).

The request handler on the remote side uses the same method to evaluate the suffix paths $U_{rel}(v_{xrpc})$ and $R_{rel}(v_{xrpc})$ using the result sequence of the function as \$ctx$_{xrpc}$ during serialization of the response message.

In case of XML data with a user-defined XML SCHEMA, the default projection algorithm is likely to throw away mandatory elements and attributes. For this reason, the runtime projection algorithm should be made schema-aware. A simple solution is to ensure that only elements with a minoccurs declaration of zero (i.e., optional elements) are removed. One can also envision more advanced variants that further reduce the size of a typed XML document.

VII. Evaluation in MonetDB/XQuery

We have implemented the proposed algorithms in *MonetDB/XQuery* [4], a purely relational XML database system that uses the *Pathfinder* [10] XQUERY compiler. We use the XRPC extension for remote function evaluation. The test platform consisted of three 2GHz Athlon64 Linux machines connected via 1Gb/s Ethernet. Each was equipped with a 2GB RAM. The benchmark data used is XMark [23], a popular XML benchmark for evaluating XQUERY efficiency and scalability. The data set was generated using scalar factors 0.1, 0.2, 0.4, 0.8 and 1.6. A data set is stored on each remote peer. We conducted three groups of experiments: bandwidth usage, query execution time and runtime projection precision. Note that, as there are no other comparative results exist, the main goal of our experiments is to show the impact of the proposed techniques in a step-by-step fashion.

We slightly modified the query Q_2^n (in Table III) so that it conforms to the XMark schema as the following:

```
(let $t := let $s := doc("xrpc://peer1/xmk_nn_MB.xml")
               /child::site/child::people/child::person
          return for $x in $s return if ($x/descendant::age < 40) then $x else ()
 return for $e in (let $c := doc("xrpc://peer2/xmk_nn_MB.auctions.xml")
                   return $c/descendant::open_auction)
        return if($c/child::seller/attribute::person = $t/attribute::id)
               then $c/child::annotation else () )/child::author
```

All techniques discussed in this paper are applied on the above query: *(i)* under the pass-by-value semantics, only the expression doc("xrpc://peer1/xmk_nn_MB.xml")/. . . /child::person can be decomposed and executed on peer1; *(ii)* under the pass-by-fragment semantics, we can decompose both the second let clause ("let \$s := ...") and the second for-loop ("for \$e in ..."), and execute them on peer1 and peer2 respectively. The variable \$t becomes the parameter of the generated function containing the second for-loop (see also Table IV); *(iii)* under the pass-by-projection semantics, the query is decomposed in the same way as using pass-by-fragment, however, when serializing the request messages, a projection of \$t/attribute::id (parameter projection) and \$c/child::annotation/child:author (result projection) is calculated. The test set thus contains four queries in total, and each of them is executed on 2 documents of sizes 10, 20, 40, 80 and 160MB.

Bandwidth usage. Figure 7 shows the bandwidth used by each benchmark query on different set of documents, i.e., the total size of XML documents plus total size of XML messages transferred among peers, in its y-axis. The x-axis is the total size of the XML documents used by each query. The pure data-shipping XQUERY query (the left most bar) costs the largest bandwidth usage, as both documents have to be shipped. By-value decomposition can push the XPATH step doc("xrpc://peer1/xmk_nn_MB.xml")/. . . /child::person to be evaluated on peer1, which reduces the amount of data sent from peer1 to the local peer. However, the second document

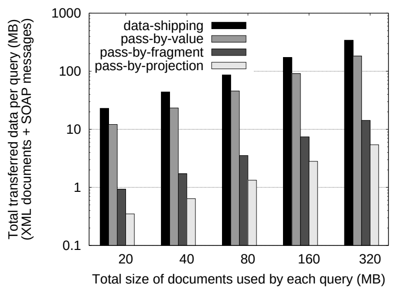


Fig. 7. Bandwidth Usage

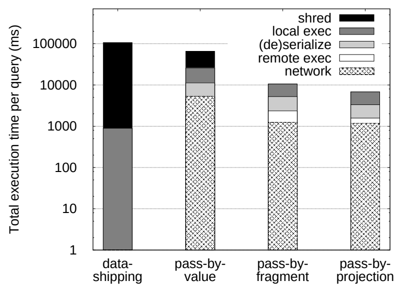


Fig. 8. Query Time Breakdown (320MB data)

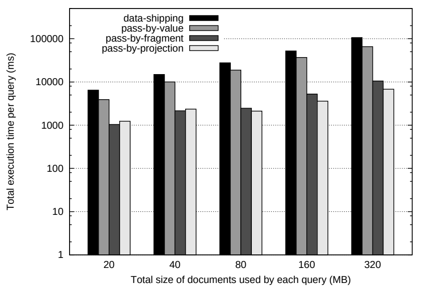


Fig. 9. Execution Time

"xmk_nn_MB.auctions.xml" still has to be sent fully. The by-fragment passing semantics allows to push predicates to both peers, achieving a distributed semijoin plan. Also, it strongly reduces message size by avoiding duplicating the same XML node multiple times. Pass-by-projection further brings down message sizes due to reduced response message size. For example, when sending the result of remote execution of the second for-loop, the response message will only contain annotation nodes with their author child nodes. In general, we observe good scalability of pass-by-fragment and pass-by-projection in bandwidth usage.

Execution time. Figure 8 shows the execution time breakdown of all four queries on documents of 320MB in total. The execution time is divided into five parts: *shred* is the time to receive a document from the remote peer and shred it in to the XML database; *local exec* is the execution time of the query at local peer, including query parsing, module loading, etc; *(de)serialize* is the time spent on generating/shredding the XML messages and extracting parameter/result values from the messages; *remote exec* is the time to execute the called functions on remote peers; and *network* is the time spent on sending/receiving the XML messages. From Figure 8, the following observations can be made: (i) in the data-shipping only query and the by-value decomposed query, data shredding is the main bottleneck, either because the whole document will be shipped (data-shipping), or an XML node might be shredded multiple times (by-value). Especially in the data-shipping query, more than 99$\%$ of the total execution time (please note the y-axis has log scale) is spent on getting the documents from remote peers and shred them; (ii) when pass-by-fragment and pass-by-projection semantics are used, the total execution times are significantly improved (about 84 $\sim$ 94$\%$, comparing with data-shipping and pass-by-value). This is easily explained as these techniques reduce the amount of data exchanged to be less than 10$\%$ of the original document sizes. Even with the overhead introduced by remote execution (i.e., '*(de)serialize*'+'*remote exec*'), pass-by-fragment or pass-by-projection are preferred over the data-shipping method. (iii) pass-by-projection performs even better than pass-by-fragment (about 35$\%$ improvement), which is again explained by the reduced bandwidth usage, as shown in Figure 7. Please note that, in these experiments we used a fast network (1Gb/s); but in a WAN environment, where much slower network performance is common, pass-by-fragment and pass-by-projection would allow queries over remote XML documents to profit even more from reduced data size.

Figure 9 shows the execution time of all queries on documents of increasing sizes, which indicates that the two enhanced parameter passing techniques achieve good scalability. Even on small documents (20MB), the proposed techniques are preferred over the data-shipping methods.

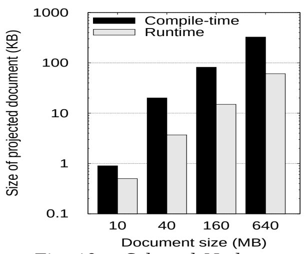


Fig. 10. Selected Nodes

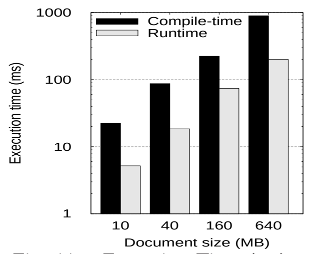


Fig. 11. Execution Time (ms)

Runtime projection precision. Our new runtime projection technique combines intermediate query results with runtime execution or relative XPATH paths. Due to selections (by e.g., predicates and value comparisons), the run-time projection node sets obtained may be much smaller than suggested by compile-time projection paths, used in [18]. We used our by-projection benchmark query to compare runtime projection with compile-time projection, on various sizes of the XMark document "xmk_nn_MB.xml". In this experiment, the compile-time technique projects all person elements and their age, while our runtime projection technique will only project those person elements that have an age descendant larger than 45. Figure 10 shows runtime projection to be 5 times more precise in terms of the size of projected document. In the case of this experiment, the investment in run-time XPATH evaluation pays off due to the more precise results, as shown in Figure 11.

VIII. Related Work

There are three main areas that are related to our proposal in this paper: distributed query processing, query decomposition and XML projection.

Much previous work in distributed query processing is surveyed in [16], [29] and parts of the book [21]. In distributed XML query processing, DXQ [9] depends on distributed query plans, in terms of the internal Galax execution algebra, generated by the Galax optimizer. In this respect, XRPC differs with its focus on interoperability, as it acts as a pure XQUERY rewriter (not making any assumptions on the system internals of the participating peers). Galax Yoo-Hoo [20] accesses web services using SOAP RPC as the communication protocol, which lacks proper support for XML elements and sequences; a problem addressed by XRPC using a specific literal SOAP message encoding. Active XML (AXML) [1], [2] is a declarative framework that harnesses web services for data integration in a peer-to-peer architecture. Like XRPC, it also used a (document/literal encoding) SOAP protocol to represent XML subtree values. However, the focus in AXML has been in adaptive call materialization strategies, not on automatic query decomposition and the semantic challenges this brings in XQUERY, such as distributed node identity. XQueryD [22], like XRPC supports function shipping in XQUERY, but it does not define an open network protocol.

Decomposing queries to address multiple data sources is by now a well-studied problem in relational databases [28] and object-oriented databases [13], [17]. Many of these ideas and methods can be applied to XQUERY, yet we have shown here that the issue of efficiently managing distributed node identity and document order add interesting challenges. [24], [25] discuss the decomposition of unstructured query languages only on a semi-structured database (a rooted, labeled graph). In XML data-bases, previous approaches require structural information about peers for supervising decomposition [27]. Other works [6], [7], [26] only focus on a restricted set of XQUERY queries.

XML projection [18] drastically reduces the size of the data model representation using compile-time query characterization. [5] introduces a precise XML pruning technique for a subset of XQUERY FLWOR expressions, based on the *apriori* knowledge of a data guide for underlying XML data. However, it does not handle XPATH predicates, backward axes and XQUERY-like languages. A type-based XML projection technique [3] is studied to improve current solutions with comparable or higher precision and less pruning overhead, as well as supporting backward XPATH axes. However, a DTD is required. [15] discusses runtime XML projection techniques. Based on the static compilation of runtime lookup-tables and a runtime-automaton from projection paths and a DTD, they can filter the input XML document efficiently using string matching algorithms. This technique, however, still lacks support for reverse XPATH axes and XQUERY built-in functions.

IX. CONCLUSION

We have described a framework for distributed execution of full-fledged XQUERY, focusing on the issue of providing equivalent query decompositions, in the face of semantic differences when (parts of) nodes are shipped across the network in XML messages. We first carefully characterized the problems that may occur regarding node identity and structural XPATH relationships in such a distributed setting. Then, we proposed a series of techniques such as pass-by-fragment and the use of a novel runtime XML projection method for serializing XML messages, that remove virtually all semantic problems and strongly improve performance, as shown by experiments on the open-source MonetDB/XQuery XML database system (monetdb.cwi.nl).

Our main future work is an issue left out-of-scope here: deciding on distributed query placement after decomposition. In this area, we also contemplate using runtime methods to improve optimization quality. Another direction is decomposition of queries containing XQUF update expressions. The challenge here is that updates are necessarily tied to execution on their source peer, which restricts decomposition to cases where at compile-time a single affected peer can be identified.

REFERENCES

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *SIGMOD*, 2004.
[2] S. Abiteboul et al. A Framework for Distributed XML Data Management. In *EDBT*, 2006.
[3] V. Benzaken et al. Type-Based XML Projection. In *VLDB*, 2006.
[4] P. Boncz et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, 2006.
[5] S. Bressan et al. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2), 2005.
[6] P. Buneman et al. Using Partial Evaluation in Distributed Query Evaluation. In *VLDB*, 2006.
[7] G. Cong et al. Distributed query evaluation with performance guarantees. In *SIGMOD*, 2007.
[8] D. Draper et al. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation 8 June 2006.
[9] M. Fernández et al. Highly Distributed XQuery with DXQ. In *SIGMOD*, 2007.
[10] T. Grust et al. XQuery on SQL Hosts. In *VLDB*, 2004.
[11] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624.
[12] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part2-20030624.
[13] V. Josifovski and T. Risch. Query decomposition for a distributed object-oriented mediator system. *Distributed and Parallel Databases*, 11(3):307–336, 2002.
[14] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. *SIGPLAN Not.*, 34(8):13–24, 1999.
[15] C. Koch et al. XML Prefiltering as a String Matching Problem. In *ICDE*, 2008.
[16] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4), 2000.
[17] H. Kozankiewicz, K. Stencel, and K. Subieta. Distributed query optimization in the stack-based approach. In *HPCC*, 2005.
[18] A. Marian et al. Projecting XML Documents. In *VLDB*, 2003.
[19] N. Mitra. SOAP Version 1.2 Part 0: Primer. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.
[20] N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, 2004.
[21] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., NJ, USA, 1999.
[22] C. Re et al. Distributed XQuery. In *IIWeb*, September 2004.
[23] A. Schmidt et al. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.
[24] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *VLDB*, 1996.
[25] D. Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1), 2002.
[26] K. Tajima and Y. Fukui. Answering XPath queries over networks by sending minimal views. In *VLDB*, 2004.
[27] L. T. T. Thuy, D. D. Duong, V. C. Bhavsar, and H. Boley. A bottom-up strategy for query decomposition. In *ICDIM*, 2006.
[28] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Trans. Database Syst.*, 1(3):223–241, 1976.
[29] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4), 1984.
[30] Y. Zhang and P. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *VLDB*, 2007.
[31] Y. Zhang and P. Boncz. Distributed XQuery and updates processing with heterogeneous XQuery engines. In *SIGMOD*, 2008.